Prepared by: G. Venusopal

Unit - IV : GREEDY TECHNIQUE

Abstracted from: Introduction to the DAA, 2nd ed, PEARSON Edy, Anany Levitin

The Greedy Technique suggests constructing a solution to an optimization Problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. on each step, the choice made must be feasible, locally optimal, and irrevocable.

- feasible means, it (choice) has to satisfy the problem's constraints
- locally optimal means, it has to be the best local choice among all feasible choices available on that step.
- irrevocable means, once made, it cannot be changed on subsequent steps of the algorithm.

Prim's algorithm Ex: change-making problem

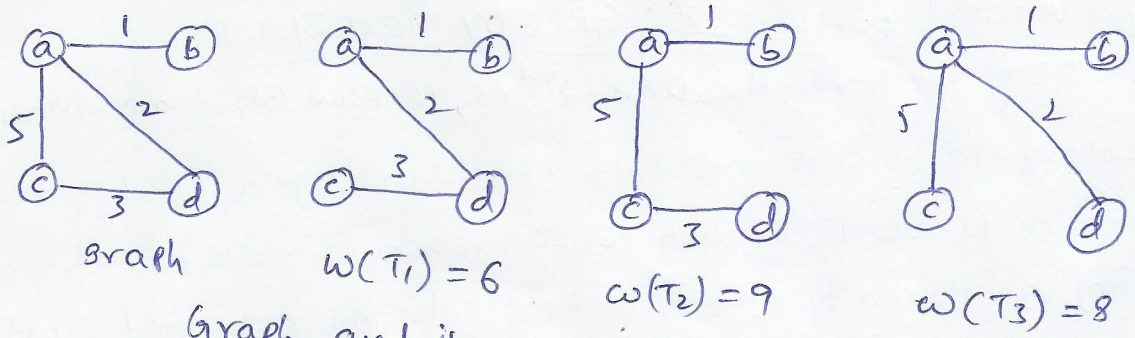
Given n points, connect them in the cheapest possible way so that there will be a path between every pair of points. we can represent the points by vertices of a graph, possible connections by the graph's edges, and the connection costs by the edge weights.

Then the problem can be defined as the minimum spanning tree. It can be defined as

definition: A spanning tree of a connected graph is its connected acyclic subgraph (i.e., tree) that contains all the vertices of the graph.

A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

So, the minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.



Graph and its spanning trees T_1, T_2, & T_3. T_1 is the MST (minimum spanning tree)

→ Exhaustive-search approach to constructing a minimum spanning tree is not suitable.

→ Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.

→ The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (A vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight).

→ The algorithm stops after all the graph's vertices have been included in the tree being constructed.

Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total no. of such iterations is $n-1$, where n is the number of vertices in the graph.

ALGORITHM Prim(G)
// Prim's algorithm for constructing a minimum spanning tree.
// Input: A weighted connected graph $G = (V, E)$.
// output: E_T, the set of edges composing a minimum spanning tree of G.
$V_T \leftarrow \{v_0\}$ // the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \phi$

for $i \leftarrow 1$ to $|V| - 1$ do

find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$.

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

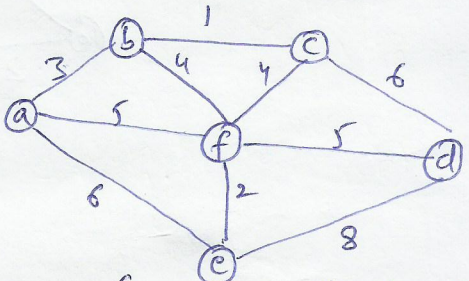
return E_T

Finding the next vertex to be added to the current tree $T = (V_T, E_T)$ becomes a simple task of finding a vertex with the smallest distance label in the set $V - V_T$.

After identifying a vertex u^* to be added to the tree, we need to perform two operations:

1. move u^* from the set $V - V_T$ to the set of tree vertices V_T.
2. For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u's current distance label, update its labels by u^* and the weight of the edge between u^* and u, respectively.

Ex: 1.



Graph. construct MST using PRIM

How efficient is Prim's algorithm?

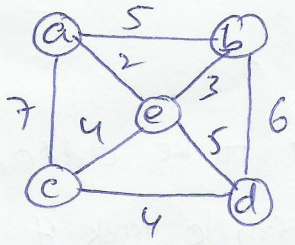
Ans: If a graph is represented by its weight matrix and the

priority queue is implemented as an unordered array, the algorithm's runtime will be $O(|V|^2)$.

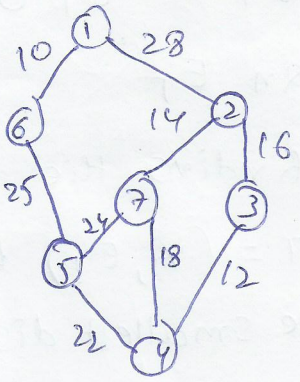
- If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the runtime of the algorithm is $O(|E| \log |V|)$.

Kruskal algorithm

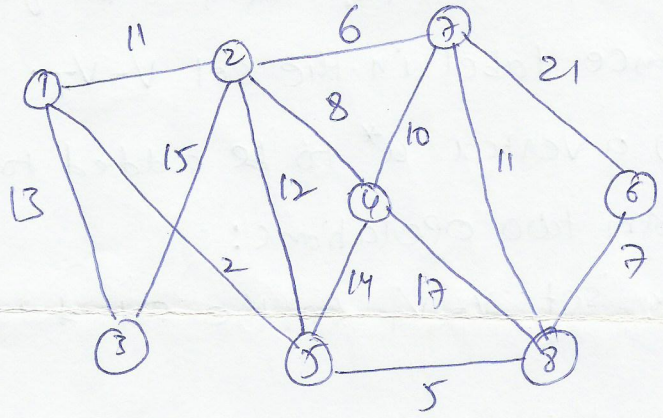
Exercises 1.



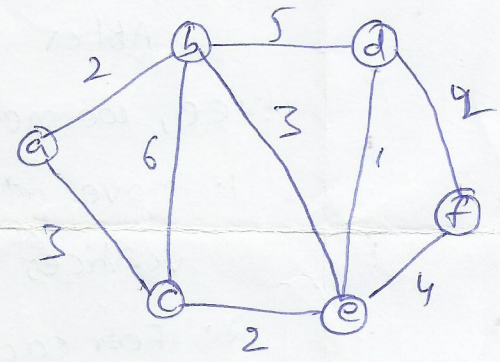
2.



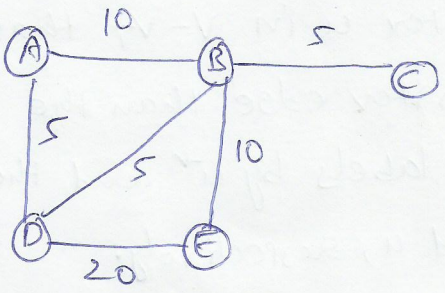
3.



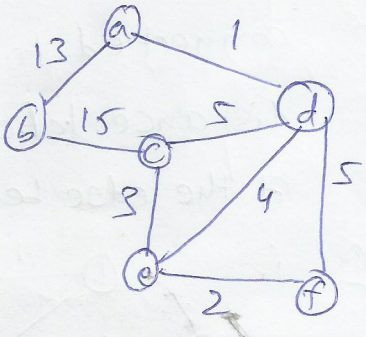
4.



5.



6.



Kruskal's algorithm

This algorithm constructs a minimum spanning tree for a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

ALGORITHM Kruskal(G)

// Kruskal's algorithm for constructing a minimum spanning tree
// Input: A weighted connected graph $G = (V, E)$.
// Output: E_T, the set of edges composing a minimum spanning tree of G.

sort E in nondecreasing order of the edge weights
$w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$.

$E_T \leftarrow \emptyset$; ecounter $\leftarrow 0$ // initialize the set of tree edges its size

$k \leftarrow 0$ // initialize the no. of processed edges

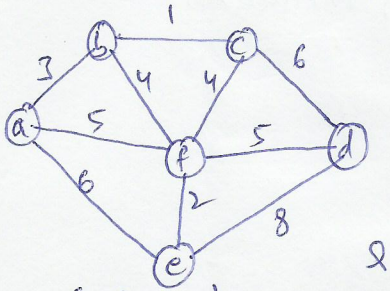
while ecounter $< |V| - 1$ do

$k \leftarrow k + 1$

if $E_T \cup \{e_{i_k}\}$ is acyclic

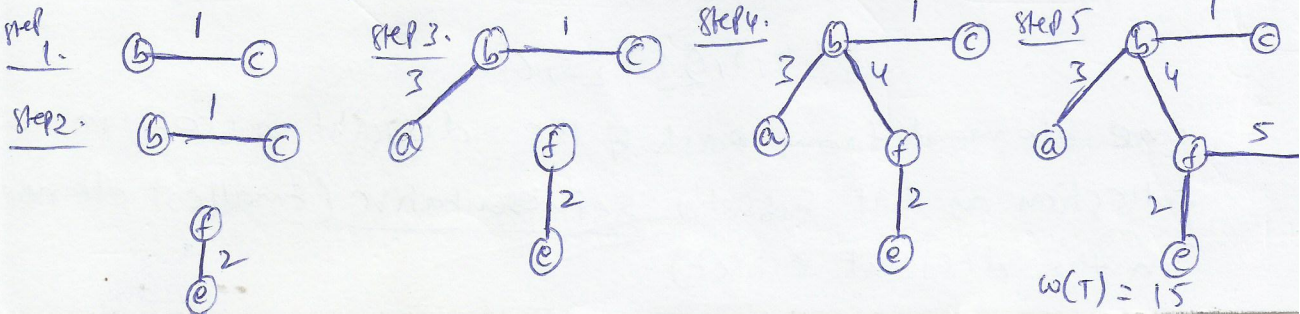
$E_T \leftarrow E_T \cup \{e_{i_k}\}$; ecounter $\leftarrow$ ecounter $+ 1$

return E_T.



$G = (V, E)$

From the previous example, sorted list of edges bc ef ab bf mf af df ae cd de
1 2 3 4 4 5 5 6 6 8



On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, if these trees are not the same, unites them in a larger tree by adding the edge (u, v). Fortunately, there are efficient algorithms for doing so, including the crucial check whether two vertices belong to the same tree. They are called union-find algorithms.

The time efficiency of Kruskal's algorithm will be $O(|E| \log |E|)$.

Disjoint subsets and union-find algorithms

operations on set S are

makeset(x) – creates a one-element set $\{x\}$. It is assumed that this operation can be applied to each of the elements of set S only once;

find(x) – returns a subset containing x;

union(x, y) – constructs the union of the disjoint subsets S_x and S_y containing x and y, respectively,

For example, let $S = \{1, 2, 3, 4, 5, 6\}$. Then makeset(i) creates the set $\{i\}$ and applying this operation six times initializes the structure to the collection of six singleton sets:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}.$$

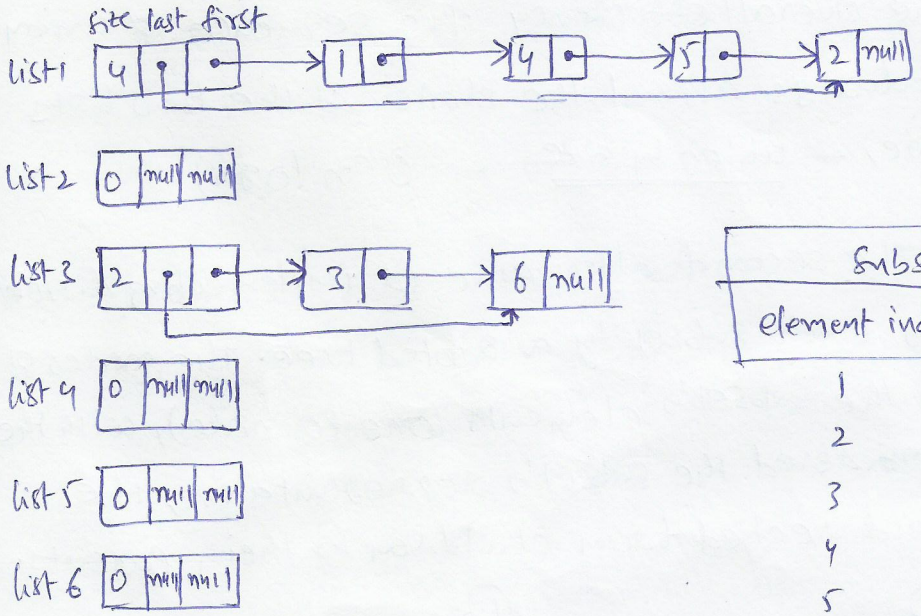
union(1,4) & union(5,2) & union(3,6) give

$$\{1,4\}, \{5,2\}, \{3,6\}.$$

and, if followed by union(4,5) & then we end up with the disjoint subsets

$$\{1,4,5,2\}, \{3,6\}.$$

– one element from each of the disjoint subsets in a collection as that subset's representative (smallest element in the disjoint subset).



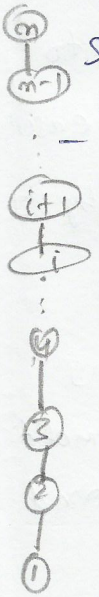
Subset representatives	
element index	representative
1	1
2	1
3	3
4	1
5	1
6	3

Linked-list representation of subsets {1,4,5,2} & {3,6} obtained by quick find after performing union(1,4), union(5,2), union(4,5) and union(3,6). The lists of size 0 are considered deleted from the collection.

There are two principal alternatives for implementing this ~~data structure~~. They are

1. quick find - optimizes the time efficiency of the find operation
2. quick union - optimizes the union operation.

- makeset(x) time efficiency is $O(1)$, and hence the initialization of n singleton subsets is in $O(n)$.
- Time efficiency of find(x) is $O(1)$; retrieve the x's representative in the representative array.
- Union(x,y) is $O(n^2)$; append y's list to the end of the x's list, update the information about their representative for all the elements in the y list, and then delete the y's list from the collection. It is to verify, however, that with this algorithm the sequence of union operations



time efficiency of

union(2,1), union(3,2), ..., union(i+1,i), ... union(n,n-1)

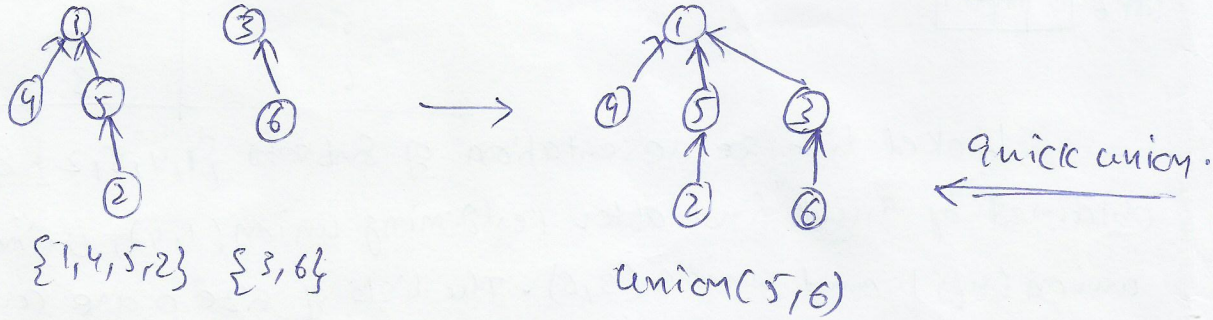
runs in $O(n^2)$ time. which is slow.

alternative1

To improve the overall efficiency of a sequence of union operations is to always append the shorter of the two lists to the longer one, — union by size. — $O(n \log n)$.

alternative2

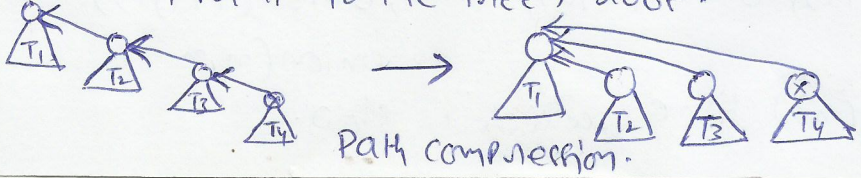
— quick union: The second alternative for implementing disjoint subsets, represents each subset by a rooted tree. The nodes of the tree contain the subset's elements (one per node), with the root's element considered the subset's representative; the tree's edges are directed from children to their parents.



A union(x,y) is implemented by attaching the root of the y's tree to the root of the x's tree. The time efficiency of this operation is clearly $O(1)$. A find(x) is performed by following the pointer chain from the node containing x to the tree's root. its time efficiency is $O(n)$ because a tree representing a subset can degenerate into a linked list with n nodes.

This time bound can be improved, Always perform a union operation by attaching a smaller tree to the root of a larger one. The size of a tree can be measured either by the number of nodes (called union by size) or by its height (called union by rank).

Better efficiency can be obtained by combining either variety of quick union with path compression. This modification makes every node encountered during the execution of a find operation point to the tree's root.



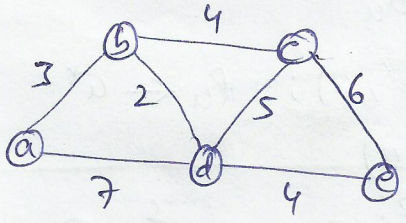
Dijkstra's Algorithm for Single-source shortest-Paths Proble

for a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertic we are not interested in a single shortest path that starts at the source and visits all the other vertices. this could have been a much more difficult problem (a version of the TSP

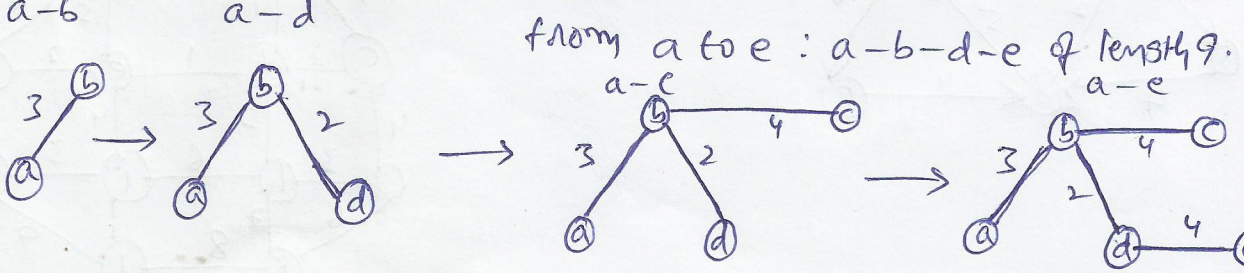
→ the single-source shortest-paths problem asks for a family of paths each leading from the source to a differe vertex in the graph, though some paths may, have edg in common.

→ The best known algorithm for solving single-source shortest-paths problem is Dijkstra's algorithm. This algorithm is applicable to graphs with nonnegative weights only.

→ Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source. First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. In general, before its ith iteration commences, the algo has already identified the shortest paths to $i-1$ other vertic nearest to the source.



The shortest paths & their leng
from a to b: a-b of length 3.
from a to d: a-b-d of length 5.
from a to c: a-b-c of length 7.
from a to e: a-b-d-e of length 9.

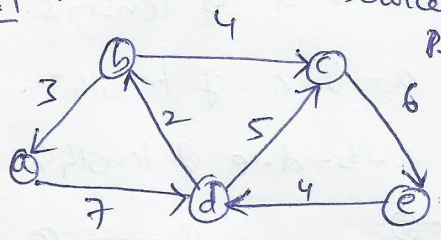


Note: Though Dijkstra's & Prim's algorithms solve different problems in different manner: Dijkstra's algorithm compares path lengths and therefore must add edge weights, while Prim's algorithm compares the edge weights as given.

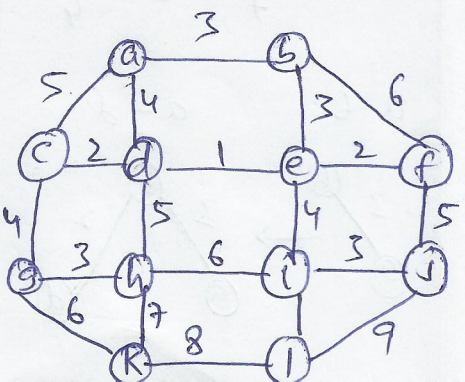
Algorithm Dijkstra(G, s)
// Dijkstra's algorithm for single-source shortest paths.
// Input: A weighted connected graph G=(V, E) with nonnegative weights and its vertex s.
// Output: The length d_v of a shortest path from s to v and its penultimate vertex P_v for every vertex v in V.

Initialize(Q) // initialize vertex priority queue to empty.
for every vertex v in V do
$d_v \leftarrow \infty$; $P_v \leftarrow$ null
Insert(Q, v, d_v) // Initialize vertex priority in the priority queue.
$d_s \leftarrow 0$; Decrease(Q, s, d_s) // update priority of s with d_s.
$V_T \leftarrow \phi$
for i ← 0 to |V| − 1 do
$u^* \leftarrow$ Deletemin(Q) // delete the minimum priority element
$V_T \leftarrow V_T \cup \{u^*\}$
for every vertex u in $V - V_T$ that is adjacent to u^* do
if $d_{u^*} + w(u^*, u) < d_u$
$d_u \leftarrow d_{u^*} + w(u^*, u)$; $P_u \leftarrow u^*$
Decrease(Q, u, d_u)

Ex:1 Solve the single source shortest path problem with vertex a as the source



Ex2:



Huffman Trees

codecoord: Assigning each text's character in an n-character alphabet to some sequence of bits is called the codecoord & encoding

There are two types of encoding. They are

(i) fixed-length encoding.

(ii) variable-length encoding.

(i) fixed-length encoding: assigns to each character a bit stri of the same length

(ii) variable-length encoding: assigns codecoords of different lens to different characters.

If we want to create a binary code for some alphab all the left edges are labeled by 0 and all the right edges a labeled by 1 (& vice versa).

Huffman algorithm

Step1. Initialize n one-bit node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's weigh (weight of the tree = sum of the frequencies in the tree's leaves

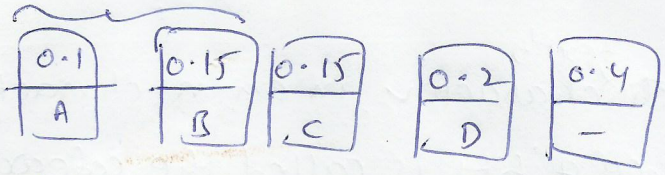
Step2. Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weigh

A tree constructed by the above algorithm is calle a Huffman tree. It defines a Huffman code.

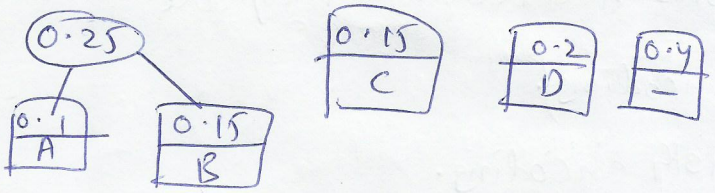
Example: Consider the five-character alphabet {A, B, C, D, _ with the following occurrence probabilities:

character	A	B	C	D	_
probability	0.1	0.15	0.15	0.2	0.4

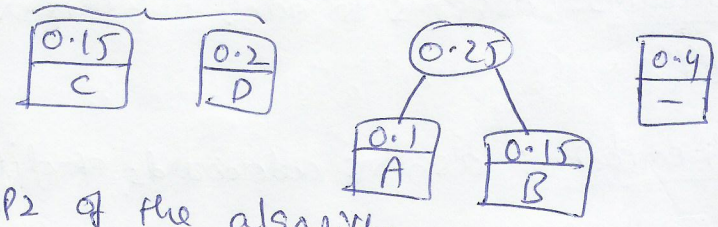
Sort the given characters according to their probabilities



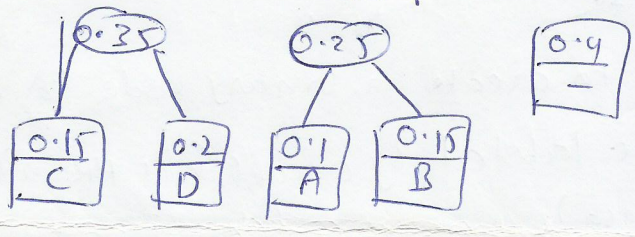
Apply step2 of the algorithm



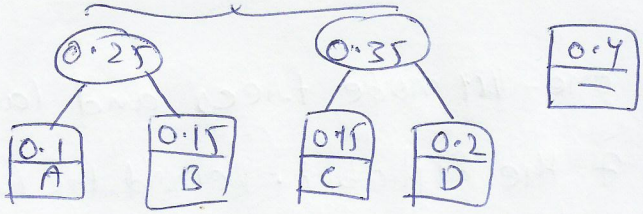
Sort the above list



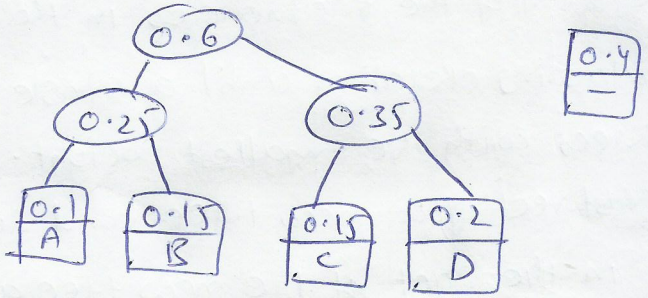
Apply step2 of the algorithm



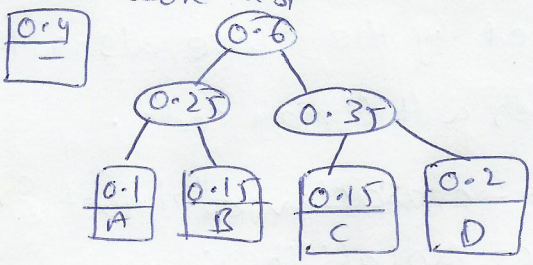
Sort the above list



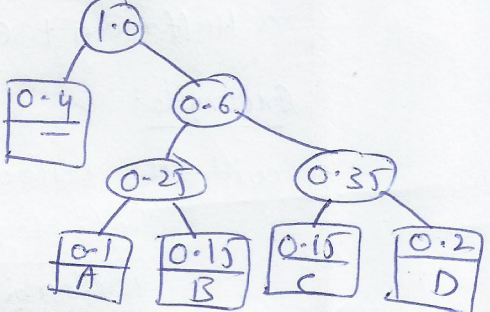
Apply step2 of the algorithm



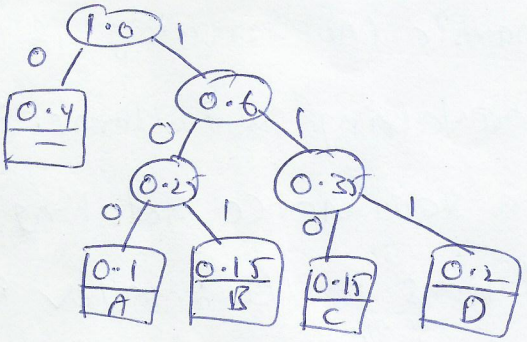
Sort the above list



Apply step2 of the algorithm, we get the final tree as



binary code for the tree to obtain Huffman code as follows



code words for A B C D _

100 101 110 111 0

Hence, DAD is encoded as 111100111, and 101110100101 is decoded as BCABD

→ The expected number of bits per character in this code

$= 3 \times 0.1 + 3 \times 0.15 + 3 \times 0.15 + 3 \times 0.2 + 1 \times 0.4$

$= 2.2$

→ Had one used a fixed-length encoding for the same alphabet, one would have to use atleast three bits per each character. Huffman's code achieves the compression ratio

$$(3 - 2.2)/3 \times 100\% = 26.67\%$$

Ex2,3,4 construct a Huffman tree for the following data and obtain its Huffman code.

(i)

character	A	B	C	D	E	_
Probability	0.5	0.35	0.5	0.1	0.4	0.2

(ii)

character	A	B	C	D	E	*
Probability	0.1	0.1	0.2	0.2	0.4	

(iii)

character	A	B	C	D	_
Probability	0.35	0.1	0.2	0.2	0.15

b. Encode the text DAD_BE (i), CAB (ii), BAD_DAB (iii)

c. Decode the text

and finally compute the compression ratio.

Dynamic Programming (DP)

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Rather than solving overlapping subproblems again and again, Dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which we can then obtain a solution to the original problem.

Ex: consider the n^{th} term of the Fibonacci sequence given by the relation:

$$F(n) = F(n-1) + F(n-2) \quad \text{when } n \geq 2$$

with initial conditions $F(0) = 0$, $F(1) = 1$

In order to compute recursively $F(4)$, the algorithm breaks the problem into two subproblems namely $F(3)$ and $F(2)$.

$$F(4) = F(3) + F(2)$$
$$F(3) = F(2) + F(1)$$
$$F(2) = F(1) + F(0)$$

Now we know that $F(2) = 1$, so this result is stored in a table. Further, in order to recursively compute $F(3) = F(2) + F($ again $F(2)$ is required to be computed, an overlapping subproblem. This time instead of computing it again, the result is looked from the table and returned.

Difference between Divide-and-conquer & Dynamic Programming (DP)

→ 1. Divide-and-conquer is best suited for the case when no overlapping sub-problems are encountered. whereas, in DP algorithms, we typically solve each subproblem only once and store their solutions.

2. In DandC recursion is possible. where as in DP, there is no recursion.

3. In DandC, the sub-problems are independent of each other while in case of DP, the sub-problems are not independent of each other.

4. In DandC, we need to solve the subproblem again and again where as in DP, we store the subproblem solution in a table (called remembering, key in DP).

Difference between Greedy & DP

Greedy	DP
1. computes the solution in top-down fashion.	1. computes the solution by making its choices in a serial fashion. (bottom-up fashion)
2. It does not use the principle of optimality.	2. It uses the principle of optimality.
3. We have a local choice of the subproblem that will lead to an optimal answer.	3. DP would solve all dependent subproblems and then select one that would lead to an optimal solution.
4. more efficient	4. less efficient.
5. cheaper.	5. Expensive.

However, there are some problems that greedy cannot solve while DP can.

Elements of DP DP is used to solve problems with the following characteristics:

1. Optimal substructure
2. overlapping subproblems

1. Optimal sub-structure (principle of optimality): An optimal solution to the problem contains within it optimal solutions to sub-problems.
2. Overlapping subproblems: there exist some places where we solve the same subproblem more than once.

matrix chain multiplication (mcm)

matrix multiplication is associative

$A_1 A_2 A_3 A_4$ can be fully paranthesized in five distinct ways:

1. $(A_1 (A_2 (A_3 A_4)))$
2. $(A_1 ((A_2 A_3) A_4))$
3. $((A_1 A_2) (A_3 A_4))$
4. $((A_1 (A_2 A_3)) A_4)$
5. $(((A_1 A_2) A_3) A_4)$

Ex: 1 $A_1 = 10 \times 100$ $A_2 = 100 \times 5$ $A_3 = 5 \times 50$ $A_4 = 50 \times 10$

no. of multiplications

$A_1 A_2 = 10 \times 100 \times 5 = 5000$ 10×5

$A_2 A_3 = 100 \times 5 \times 50 = 25000$ 100×50

$A_3 A_4 = 5 \times 50 \times 10 = 2500$ 5×10

$A_2 (A_3 A_4) = 100 \times 5 \times 10 = 5000$ 100×10

$A_1 (A_2 A_3) = 10 \times 100 \times 50 = 50,000$ 10×50

$(A_1 A_2) A_3 = 10 \times 5 \times 50 = 2500$ 10×50

$(A_2 A_3) A_4 = 100 \times 50 \times 10 = 5,000$ 100×10

$(A_1 (A_2 (A_3 A_4))) = 10 \times 100 \times 10 = 10,000$ 10×10

$(A_1 ((A_2 A_3) A_4)) = 10 \times 100 \times 10 = 10,000$ 10×10

$((A_1 A_2) (A_3 A_4)) = 10 \times 5 \times 10 = 500$ 10×10 ✓

$((A_1 (A_2 A_3)) A_4) = 10 \times 50 \times 10 = 5,000$ 10×10

$(((A_1 A_2) A_3) A_4) = 10 \times 50 \times 10 = 5,000$ 10×10

Ex: 2 $A_1 = 2 \times 3$ $A_2 = 3 \times 4$ $A_3 = 4 \times 5$

$(A_1 A_2) A_3$ $A_1 A_1 = 2 \times 3 \times 4 = 24$ 2×4 $(A_1 A_2) A_3 = 2 \times 4 \times 5 = 40$ 2×5

$A_1 (A_2 A_3)$ $A_2 A_3 = 3 \times 4 \times 5 = 60$ 3×5 $A_1 (A_2 A_3) = 2 \times 3 \times 5 = 30$ 2×5

$A_1 A_2$

```
MATRIX-CHAIN-ORDER(P)
  n ← length[P] - 1
  for i ← 1 to n
    do m[i,i] ← 0
  for l ← 2 to n    // l is the chain length.
    do for i ← 1 to n-l+1.
        do j ← i+l-1.
           m[i,j] ← ∞
           for k ← i to j-1
             do q ← m[i,k] + m[k+1, j] + p_{i-1} p_k p_j
                if q < m[i,j]
                  then m[i,j] ← q
                       s[i,j] ← k
  return m and s.
```

The initial call PRINT-OPTIMAL-PARENS(s, i, j)

```
  if i == j
    then print "A_i".
    else print "("
         PRINT-OPTIMAL-PARENS(s, i, s[i,j])
         PRINT-OPTIMAL-PARENS(s, s[i,j]+1, j)
         Print ")".
```

Recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if} \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if} \end{cases}$$

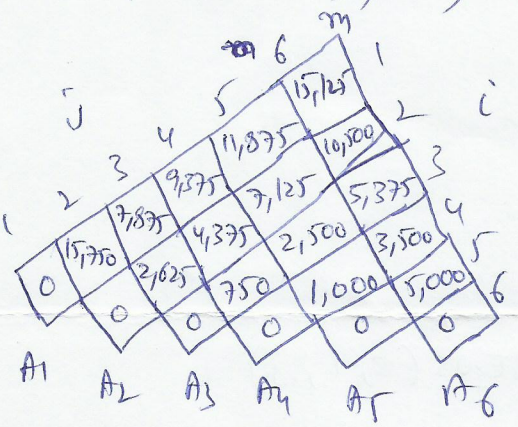
matrix	dimension
A1	30 X 35
A2	35 X 15
A3	15 X 5
A4	5 X 10
A5	10 X 20
A6	20 X 25

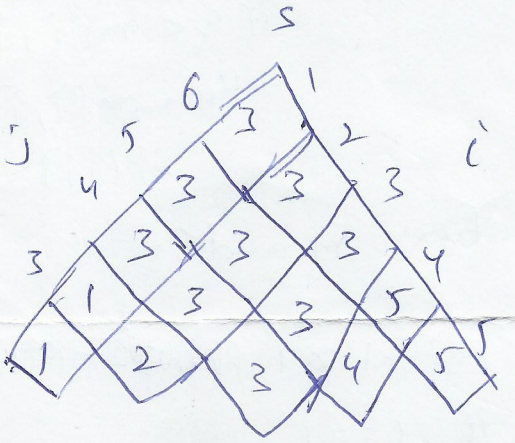
$n = 6$

$A_i \longrightarrow P_{i-1} \times P_i$

1. $((A_1 A_2)((A_3 A_4)(A_5 A_6)))$
2. $(((A_1 A_2)(A_3 A_4))(A_5 A_6))$
3. $(A_1(A_2(A_3(A_4(A_5 A_6)))))$
4. $((((((A_1 A_2) A_3) A_4) A_5) A_6)$
5. $(A_1(((A_2(A_3 A_4)) A_5) A_6))$

.....





m and s tables computed by MATRIX-CHAIN-ORDER for n

$P_0 - 30 \quad P_1 - 35 \quad P_2 - 15 \quad P_3 - 5 \quad P_4 - 10 \quad P_5 - 20 \quad P_6 - 25$

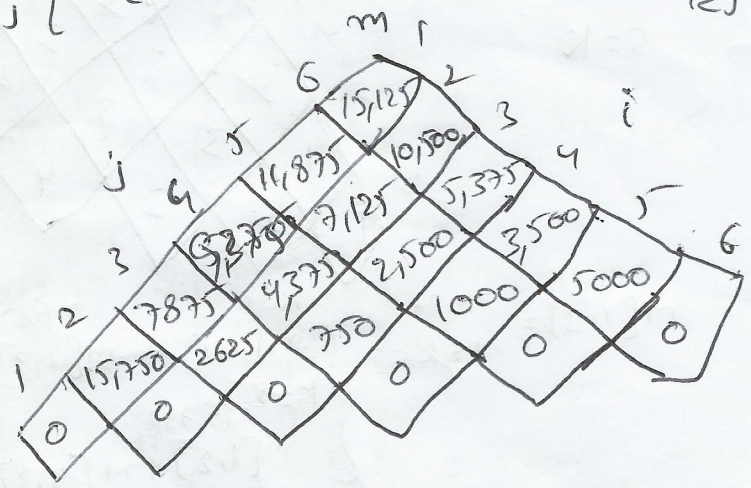
$$m[1,3] = \min_{1 \le k < 3} \begin{cases} k=1 & 0 \quad 2,625 \quad 30 \cdot 35 \cdot 5 \\ m[1,1] + m[2,3] + P_0 P_1 P_3 = 7875 \checkmark \\ k=2 \\ m[1,2] + m[3,3] + P_0 \cdot P_2 \cdot P_3 = 18,000 \\ 15,750 + 0 \quad + 30 \cdot 15 \cdot 5 \end{cases}$$

$s[1,3] = k$ that optimizes $m[1,3]$

Matrix chain multiplication (9)

$$m[i,j] = \min_{i \le k < j} \begin{cases} 0 & \text{if } i = j \\ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j & \text{if } i < j \end{cases}$$

matrix	dimension
A_1	30 × 35
A_2	35 × 15
A_3	15 × 5
A_4	5 × 10
A_5	10 × 20
A_6	20 × 25



$m[i,j] = 0$ if $i = j$

	p_0	p_1	p_2	p_3	p_4	p_5	p_6
	30	35	15	5	10	20	25

$$m[1,2] = \min_{k=1} \; m[1,1] + m[2,2] + p_0 p_1 p_2$$

$$= 0 + 0 + 30 \cdot 35 \cdot 15 = 15{,}750$$

$$m[2,3] = \min_{k=2} \; m[2,2] + m[3,3] + p_1 p_2 p_3$$

$$= 0 + 0 + 35 \cdot 15 \cdot 5 = 2{,}625$$

35×15 → 2

15×5 → 1

35×5

$35 \times 5 + 35 \times 15 + 15 \times 5$

$175 + 525 + 75$

2625

$$m[3,4] = \min_{k=3} \; m[3,3] + m[4,4] + p_2 p_3 p_4$$

$$= 0 + 0 + 15 \cdot 5 \cdot 10 = 750$$

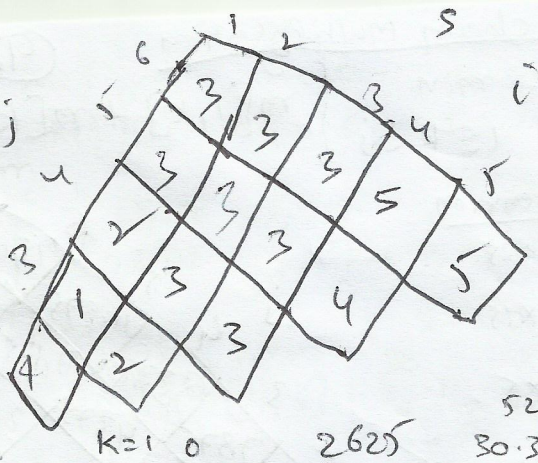
$$m[4,5] = \min_{k=4} \; m[4,4] + m[5,5] + p_3 p_4 p_5$$

$$= 0 + 0 + 5 \cdot 10 \cdot 20 = 1000$$

$$m[5,6] = \min_{k=5} \; m[5,5] + m[6,6] + p_4 p_5 p_6$$

$$= 0 + 0 + 10 \cdot 20 \cdot 25 = 5000$$

$s = k$



$$m[1,3] = \min_{1 \le k < 3} \begin{cases} k=1: & m[1,1] + m[2,3] + p_0 p_1 p_3 = 7875 \checkmark \\ k=2: & m[1,2] + m[3,3] + p_0 p_2 p_3 = 18{,}000 \end{cases}$$

$$m[2,4] = \min_{2 \le k < 4} \begin{cases} k=2: & m[2,2] + m[3,4] + p_1 p_2 p_4 = 6000 \\ k=3: & m[2,3] + m[4,4] + p_1 p_3 p_4 = 4375 \checkmark \end{cases}$$

$$m[3,5] = \min_{3 \le k < 5} \begin{cases} k=3: & m[3,3] + m[4,5] + p_2 p_3 p_5 = 2500 \checkmark \\ k=4: & m[3,4] + m[5,5] + p_2 p_4 p_5 = 3750 \end{cases}$$

$$m[4,6] = \min_{4 \le k < 6} \begin{cases} k=4: & m[4,4] + m[5,6] + p_3 p_4 p_6 = 6250 \\ k=5: & m[4,5] + m[6,6] + p_3 p_5 p_6 = 3500 \checkmark \end{cases}$$

$$m[1,4] = \min_{1 \le k < 4} \begin{cases} k=1: & m[1,1] + m[2,4] + p_0 p_1 p_4 = 14825 \\ k=2: & m[1,2] + m[3,4] + p_0 p_2 p_4 = 25050 \\ k=3: & m[1,3] + m[4,4] + p_0 p_3 p_4 = 9375 \checkmark \end{cases}$$

$$m[2,5] = \min_{2 \le k < 5} \begin{cases} k=2 \\ m[2,2] + m[3,5] + p_1 p_2 p_5 \\ \quad 0 \quad 2{,}500 \quad 35 \cdot 15 \cdot 20 = 10500 \quad = 13{,}000 \\ k=3 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 \\ \quad 2625 \quad 1000 \quad 35\ 5\ 20 = 3500 \quad = 7125 \checkmark \\ k=4 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 \\ \quad 4375 \quad 0 \quad 35\ 10\ 20 = 7000 \quad = 11{,}375 \end{cases}$$

$$m[3,6] = \min_{3 \le k < 6} \begin{cases} k=3 \\ m[3,3] + m[4,6] + p_2 p_3 p_6 \\ \quad 0 \quad 3{,}500 \quad 15\ 5\ 25 = 1875 \quad = 5375 \checkmark \\ k=4 \\ m[3,4] + m[5,6] + p_2 p_4 p_6 \\ \quad 750 \quad 5000 \quad 15\ 10\ 25 = 3750 \quad = 8750 \\ k=5 \\ m[3,5] + m[6,6] + p_2 p_5 p_6 \\ \quad 2500 \quad 0 \quad 15\ 20\ 25 = 7500 \quad = 10{,}000 \end{cases}$$

$$m[1,5] = \min_{1 \le k < 5} \begin{cases} k=1 \\ m[1,1] + m[2,5] + p_0 p_1 p_5 \\ \quad 0 \quad 7125 \quad 30\ 35\ 20 = 21000 \quad = 28125 \\ k=2 \\ m[1,2] + m[3,5] + p_0 p_2 p_5 \\ \quad 15{,}750 \quad 2{,}500 \quad 30\ 15\ 20 = 9000 \quad = 27250 \\ k=3 \\ m[1,3] + m[4,5] + p_0 p_3 p_5 \\ \quad 7{,}875 \quad 1000 \quad 30\ 5\ 20 = 3000 \quad = 11{,}875 \checkmark \\ k=4 \\ m[1,4] + m[5,5] + p_0 p_4 p_5 \\ \quad \text{(illegible)} \quad 0 \quad 30\ 10\ 20 = 6000 \quad = 15{,}375 \times \end{cases}$$

$$m[2,6] = \min_{2 \le k < 6} \begin{cases} k=2 \\ m[2,2] + m[3,6] + p_1 p_2 p_6 \\ \quad 0 \quad 5375 \quad 35\ 15\ 25 = 13125 \quad = 18{,}500 \\ \quad 375 \times 35 = 1875,\ 1875 + 11250 = 13125 \\ k=3 \\ m[2,3] + m[4,6] + p_1 p_3 p_6 \\ \quad 2625 \quad 3500 \quad 35\ 5\ 25 = 4375 \quad = 10{,}500 \checkmark \\ \quad 125 \times 35 = 4375 \\ k=4 \\ m[2,4] + m[5,6] + p_1 p_4 p_6 \\ \quad 4375 \quad 5000 \quad 35\ 10\ 25 = 8750 \quad = 18125 \\ k=5 \\ m[2,5] + m[6,6] + p_1 p_5 p_6 \\ \quad 7125 \quad 0 \quad 35\ 20\ 25 = 17500 \quad = 24625 \end{cases}$$

$12 \times 35 = 375$

$$m[1,6] = \min_{1 \le k < 6} \begin{cases} k=1 & \overset{0}{m[1,1]} + \overset{10,500}{m[2,6]} + \overset{2625}{\underset{30\ 35\ 25}{P_0\ P_1\ P_6}} \\ k=2 & \overset{15,750}{m[1,2]} + \overset{5,375}{m[3,6]} + \overset{11250}{\underset{30\ 15\ 25}{P_0\ P_2\ P_6}} \\ k=3 & \overset{7875}{m[1,3]} + \overset{3,500}{m[4,6]} + \overset{3750}{\underset{30\ 5\ 25}{P_0\ P_3\ P_6}} = 15,125 \checkmark \\ k=4 & \overset{9,375}{m[1,4]} + \overset{5,000}{m[5,6]} + \overset{7500}{\underset{30\ 10\ 25}{P_0\ P_4\ P_6}} \\ k=5 & \overset{11,875}{m[1,5]} + \overset{0}{m[6,6]} + \overset{15000}{\underset{30\ 20\ 25}{P_0\ P_5\ P_6}} \end{cases}$$

$s[1,6] = 3$

$(A_1\ A_2 A_3)\ (A_4\ A_5\ A_6)$

$(A_1\ (A_2 A_3))\ ((A_4 A_5)\ A_6)$

The Knapsack Problem and memory Functions

Problem: Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

To obtain optimal solution, the recurrence we have to use is as follows

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

and the initial conditions are

$$V[0,j] = 0 \quad \text{for } j \geq 0 \quad \&$$
$$V[i,0] = 0 \quad \text{for } i \geq 0$$

Our goal is to find $V[n,W]$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

For $i,j > 0$, to compute the entry in the i^{th} row and the j^{th} column, $V[i,j]$, we compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left.

Apply Initial conditions

→ Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity W = 5

Solve the instance of the knapsack Problem by DP algorithm

Apply initial conditions first

$v[0,0] = 0$, $v[0,1] = 0$, $v[0,2] = 0$, $v[0,3] = 0$, $v[0,4] = 0$,

$v[0,5] = 0$

$v[1,0] = 0$, $v[2,0] = 0$, $v[3,0] = 0$, $v[4,0] = 0$

capacity j

i	0	1	2	3	4	5
0	0	0	0	0	0	0
$\omega_1=2, v_1=12$ 1	0	0	12	12	12	12
$\omega_2=1, v_2=10$ 2	0	10	12	22	22	22
$\omega_3=3, v_3=20$ 3	0	10	12	22	30	32
$\omega_4=2, v_4=15$ 4	0	10	15	25	30	37

$v[1,1] = v[0,1] = 0$, $j - \omega_i = 1-2 < 0$

$v[1,2] = \max\{v[0,2], 12 + v[0,0]\}$ $j - \omega_i = 2-2 = 0$

$= 12$

$v[1,3] = \max\{v[0,3], 12 + v[0,1]\} = 12$, $j - \omega_i = 3-2 = 1 > 0$

$v[1,4] = 12$ $v[1,5] = 12$

$v[2,1] = \max\{v[1,1], 10 + v[1,0]\}$, $j - \omega_i = 1-1 = 0$

$= 10$

$v[2,2] = \max\{v[1,2], 10 + v[1,1]\} = 12$, $j - \omega_i = 2-1 = 1 > 0$

$v[2,3] = \max\{v[1,3], 10 + v[1,2]\} = 22$,

$v[2,4] = \max\{v[1,4], 10 + v[1,3]\} = 22$, $v[2,5] = 22$

$v[3,1] = v[2,1] = 10$ $j - \omega_i = 1-3 < 0$

$v[3,2] = v[2,2] = 12$ $j - \omega_i = 2-3 < 0$

$v[3,3] = \max\{v[2,3], 20 + v[2,0]\} = 22$, $j - \omega_i = 3-3 = 0$

$v[3,4] = \max\{v[2,4], 20 + v[2,1]\} = 30$,

$v[3,5] = \max\{v[2,5], 20 + v[2,2]\} = 32$,

$V[4,1] = V[3,1] = 10$, $\quad j - w_i = 1 - 2 < 0$

$V[4,2] = \max\{V[3,2], 15 + V[3,0]\} = 15$,
(12) (0)

$V[4,3] = \max\{V[3,3], 15 + V[3,1]\} = 25$,
(22) (10)

$V[4,4] = \max\{V[3,4], 15 + V[3,2]\} = 30$,
(30) (12)

$V[4,5] = \max\{V[3,5], 15 + V[3,3]\} = 37$.
(32) (22)

Thus, the maximal value is $V[4,5] = \$37$.

$V[i,j] \neq V[i,j]$ [4,5] [3,5], item 4 was included in an optimal solution.

$w_4 = 2$, $\quad j - w_4 = 5 - 2 = 3$ remaining knapsack capacity.

$V[3,3] = V[2,3]$, item 3 not included.

$V[2,3] \neq V[1,3]$, item 2 was included in an optimal solution

$w_2 = 1$, $\quad j - w_2 = 3 - 1 = 2$ remaining knapsack capacity.

$V[1,2] \neq V[0,2]$, item 1 was included in an optimal solution

Hence, the optimal solution is {item 1, item 2, item 4} and value is 37.

The time & space efficiency is $O(nW)$.

Memory Functions

The goal is to set a method that solves only subproblems which are necessary and does it only once. Such a method exists; it is based on using memory functions.

Initialize the table entries with -1's except for row 0 and column 0 which are initialized with 0's. After initializing the table, the recursive function needs to be called

with $i=n$ (the no. of items) and $j=W$ (the capacity of the knapsack).

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2, v_1=12$	1	0	-1	-1	-1	-1	-1
$w_2=1, v_2=10$	2	0	-1	-1	-1	-1	-1
$w_3=3, v_3=20$	3	0	-1	-1	-1	-1	-1
$w_4=2, v_4=15$	4	0	-1	-	-	-1	-1

Algorithm MFKnapsack(i, j)
// Implements the memory function method for the knapsack problem
if V[i, j] < 0
    if j < weights[i]
        value ← MFKnapsack(i-1, j)
    else
        value ← max(MFKnapsack(i-1, j), values[i] + MFKnapsack(i-1, j-weights[i]))
    V[i, j] ← value
return V[i, j].

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2, v_1=12$	1	0	0	12	12	12	12
$w_2=1, v_2=10$	2	0	-	12	22	-	22
$w_3=3, v_3=20$	3	0	-	-	22	-	32
$w_4=2, v_4=15$	4	0	-	-	-	-	37

Example of solving an instance of the knapsack problem by the memory function algorithm.

→ Apply the bottom-up DP algorithm to the following instance of the knapsack problem:

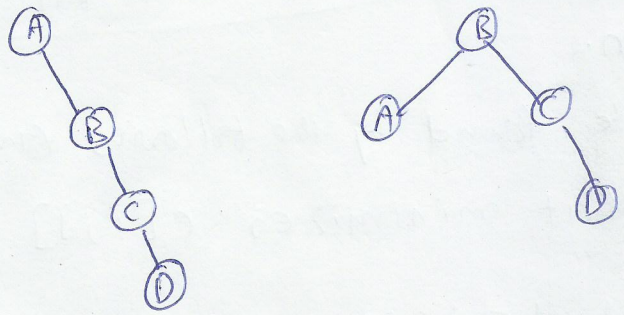
item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

capacity $W = 10$

Optimal binary search tree (OBST)

A binary search tree is used to implement a dictionary, a set of elements with the operations of search, insertion and deletion. If the probabilities of searching for elements of a set are known.

For optimal binary search tree (OBST) – the average no. comparisons in a search is the smallest possible.



Two out of 14 possible BSTs with keys A, B, C, & D which is optimal?

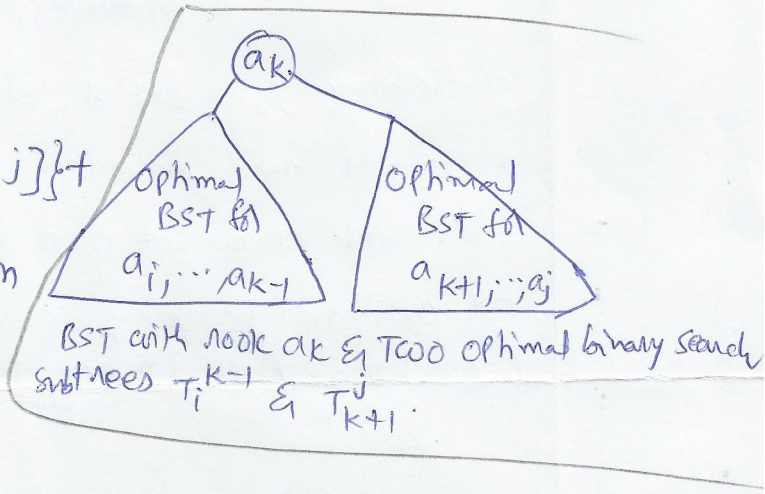
Consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively.

→ the average no. of comparisons in a successful search in first tree is $0.1\times1 + 0.2\times2 + 0.4\times3 + 0.3\times4 = 2.9$

second tree is $0.1\times2 + 0.2\times1 + 0.4\times2 + 0.3\times3 = 2.1$.

Neither of these two trees is, optimal.

let $a_1, \cdots, a_n$ be distinct keys ordered from the smallest to the largest and let $p_1, \cdots, p_n$ be the probabilities of searching for them. Let $c[i, j]$ be the smallest average no. of comparisons made in a successful search in a BST T_i^j made up of keys $a_i, \cdots, a_j$, where i, j are some integer indices, $1 \le i \le j \le n$. For such a BST, the root contains key a_k, the left subtree T_i^{k-1} contains keys $a_i, \cdots, a_{k-1}$ optimally arranged, and the right subtree T_{k+1}^j contains keys $a_{k+1}, \cdots, a_j$ also optimally arranged.



Thus, we have the recurrence

$$c[i, j] = \min_{i \le k \le j} \{c[i, k-1] + c[k+1, j]\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n$$

$$c[i, i-1] = 0 \quad \text{for } 1 \le i \le n+1$$

$$c[i, i] = p_i \quad \text{for } 1 \le i \le n.$$

Root of the tree can be found by the following formula:

$R[i, j] = k$ that minimizes $c[i, j]$

$R[i, i] = i$ for $1 \le i \le n$.

→ Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0				
2		0			
3			0		
4				0	
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

$c[1,0] = c[2,1] = c[3,2] = c[4,3] = c[5,4] = 0$

$c[1,1] = P_1 = 0.1 \quad c[2,2] = P_2 = 0.2 \quad c[3,3] = P_3 = 0.4$

$c[4,4] = P_4 = 0.3$

$R[1,1] = i = 1 \quad R[2,2] = 2 \quad R[3,3] = 3 \quad R[4,4] = 4.$

$$c[1,2] = \min \begin{cases} k=1: \overset{0}{c[1,0]} + \overset{0.2}{c[2,2]} + \sum_{s=1}^{2} P_s = 0+0.2+0.3 = 0.5 \\ k=2: \underset{0.1}{c[1,1]} + \underset{0}{c[3,2]} + \sum_{s=1}^{2} P_s = 0.1+0+0.3 = 0.4 \checkmark \end{cases}$$

$R[1,2] = 2$

$$c[2,3] = \min \begin{cases} k=2: \overset{0}{c[2,1]} + \overset{0.4}{c[3,3]} + \sum_{s=2}^{3} P_s = 0.4+0.6 = 1.0 \\ k=3: \underset{0.2}{c[2,2]} + \underset{0}{c[4,3]} + \sum_{s=2}^{3} P_s = 0.2+0.6 = 0.8 \checkmark \end{cases}$$

$R[2,3] = 3$

$$c[3,4] = \min \begin{cases} k=3: \overset{0}{c[3,2]} + \overset{0.3}{c[4,4]} + \sum_{s=3}^{4} P_s = 0.3+0.7 = 1.0 \checkmark \\ k=4: \underset{0.4}{c[3,3]} + \underset{0}{c[5,4]} + \sum_{s=3}^{4} P_s = 0.4+0.7 = 1.1 \end{cases}$$

$R[3,4] = 3$

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

$$c[1,3] = \min \begin{cases} k=1: \overset{0}{c[1,0]} + \overset{0.8}{c[2,3]} + \sum_{s=1}^{3} P_s = 0.8+0.7 = 1.5 \\ k=2: \overset{0.2}{c[1,1]} + \overset{0.4}{c[3,3]} + \sum_{s=1}^{3} P_s = 0.6+0.7 = 1.3 \\ k=3: \overset{0.4}{c[1,2]} + \overset{0}{c[4,3]} + \sum_{s=1}^{3} P_s = 0.4+0.7 = 1.1 \checkmark \end{cases}$$

$R[1,3] = 3$

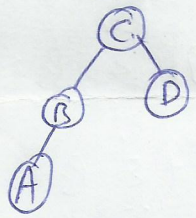
$$c[2,4] = \min \begin{cases} k=2: c[2,1] + c[3,4] + \sum_{s=2}^{4} P_s = 1.0 + 0.9 = 1.9 \\ k=3: c[2,2] + c[4,4] + \sum_{s=2}^{4} P_s = 0.5 + 0.9 = 1.4 \checkmark \\ k=4: c[2,3] + c[5,4] + \sum_{s=2}^{4} P_s = 0.8 + 0.9 = 1.7 \end{cases}$$

$R[2,4] = 3$

$$c[1,4] = \min \begin{cases} k=1: c[1,0] + c[2,4] + \sum_{s=1}^{4} P_s = 1.4 + 1.0 = 2.4 \\ k=2: c[1,1] + c[3,4] + \sum_{s=1}^{4} P_s = 0.1 + 1.0 + 1.0 = 2.1 \\ k=3: c[1,2] + c[4,4] + \sum_{s=1}^{4} P_s = 0.4 + 0.3 + 1.0 = 1.7 \checkmark \\ k=4: c[1,3] + c[5,4] + \sum_{s=1}^{4} P_s = 1.1 + 1.0 = 2.1 \end{cases}$$

$R[1,4] = 3$ is root of the tree. i.e., third key is root C.

Its left & right subtrees made up of keys A, B & D



OBST

since $R[1,2] = 2$, the root of the optimal tree containing A & B is B, with A being its left child.

→ Let the identifier set (a_1, a_2, a_3, a_4) = (cout, float, if, while) with probabilities $P_1 = 1/20$, $P_2 = 1/5$, $P_3 = 1/10$, $P_4 = 1/20$. Using the $r(i,j)$'s, construct the OBST.

→ let $n = 4$ & (a_1, a_2, a_3, a_4) = (do, if, int, while). Let $P(1:4) = (3, 3, 1, 1)$. Using the $r(i,j)$'s, construct the OBST.

Algorithm OptimalBST(P[1...n])
// Finds an optimal binary search tree by dynamic Programming

```
for i ← 1 to n do
    C[i, i-1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n+1, n] ← 0
for d ← 1 to n-1 do    // diagonal count
    for i ← 1 to n-d do
        j ← i+d
        minval ← ∞
        for k ← i to j do
            if C[i, k-1] + C[k+1, j] < minval
                minval ← C[i, k-1] + C[k+1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i+1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

Time efficiency is cubic i.e., $O(n^3)$.

space efficiency is quadratic i.e., $O(n^2)$.

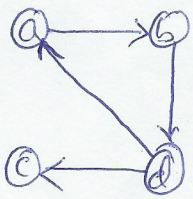
Warshall's and Floyd's Algorithms

Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem.

Warshall's Algorithm

Adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its i^{th} row and j^{th} column if and only if there is a directed edge from the i^{th} vertex to the j^{th} vertex.

Definition: The transitive closure of a directed graph with n vertices can be defined as the n-by-n boolean matrix $T = \{t_{ij}\}$, in which the element in the i^{th} row ($1 \le i \le n$) and the j^{th} column ($1 \le j \le n$) is 1 if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0.



(a) Digraph

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b) Its adjacency matrix

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c) Its transitive closure.

Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n-by-n boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}.$$

R^{0} – which does not allow any intermediate vertices in its paths; adjacency matrix of the digraph.

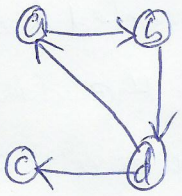
$R^{(1)}$ – contains the info about paths that can use the first vertex as intermediate;

⋮

$R^{(n)}$ – reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing else but the digraph's transitive closure.

formula for generating the elements of $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ (or) } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right).$$



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

ones reflect the existence of paths with no intermediate vertices; boxed row & column are used for getting $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex a.
note: new path from d to b.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

intermediate vertices (ivs) numbered not higher than 2, i.e. a & b
note: a → d & d → d are new paths

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

ivs numbered not higher than 3, i.e. a, b, & c
note: no new paths

$R^{(4)}$ =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

ivs numbered not higher than 4, i.e., a, b, c & d.
note: five new paths.

ALGORITHM warshall($A[1\ldots n, 1\ldots n]$)

$R^{(0)} \leftarrow A$

for $k \leftarrow 1$ to n do

  for $i \leftarrow 1$ to n do

    for $j \leftarrow 1$ to n do

      $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ & $R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$

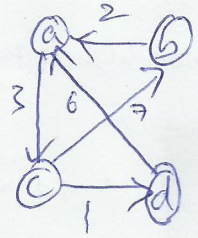
return $R^{(n)}$.

Time efficiency is $\Theta(n^3)$.

Floyd's Algorithm for the All-Pairs shortest-Paths problem

Given a weighted connected graph (undirected & directed), the all-pairs shortest-paths problem asks to find the distances (the lengths of the shortest paths) from each vertex to all other vertices.

Distance matrix: the element dij in the i^{th} row and the j^{th} column of this matrix indicates the length of the shortest path from the ith vertex to the j^{th} vertex ($1 \le i, j \le n$).



(a) Digraph

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

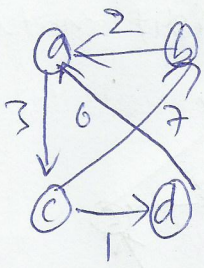
(b) Its weight matrix

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c) Its distance matrix.

Floyd's is similar to warshall. It computes the distance matrix of weighted graph with n vertices through a series of n-by-n matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots D^{(n)}.$$



$D^{(0)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices (simply the weight matrix).

$D^{(1)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices (ivs) numbered not higher than 1, i.e just a
note: b → c & d → c are new paths.

$D^{(2)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

with ivs numbered not higher th
2, i.e., a & b.
note: c → a, new path.

$D^{(3)}$ =

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

with ivs numbered not higher tha
3, i.e., a, b & c.
note: a → b, a → d, b → d & d → b ar
new paths.

$D^{(4)}$ =

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

with ivs numbered not higher than 4,
i.e., a, b, c & d.
note: c → a, new path.

Algorithm Floyd(W[1..n, 1..n])

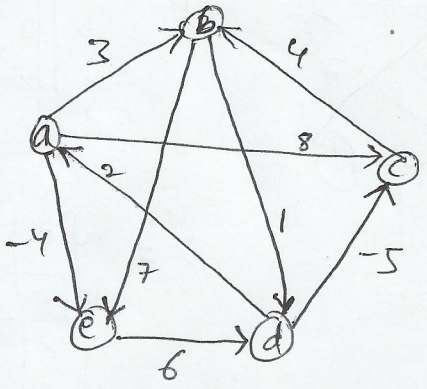
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      D[i, j] ← min{ D[i, j], D[i, k] + D[k, j] }
return D

Time efficiency is cubic i.e $\Theta(n^3)$.

(1)

→ Find the Distance matrix from the following weight matrix.

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}$$



Solution:

$$D = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$